Spearman rank correlation between the user's ratings and general ratings or sales volume; other items are ignored. Other embodiments perform rank correlation based on this restricted sample and separately perform rank correlation upon all items rated by the user, and perform a weighted average on the results.

Note 1: In some embodiments, it is possible for a user to change his review and rating of an item over time, since he may come to feel differently about it with more experience. But for purposes of calculating, his earlier ratings are stored. In preferred such iterations, the last rating of an item entered on the first day that he rated that item is used.

Note 2: In cases where the cluster has too few ratings or sales to do meaningful calculations, "virtual" clusters can be created by combining clusters with similar taste signatures into one larger clusters for purpose of computing representativeness. In preferred such embodiments, clusters are successively added to the original cluster, and the representativeness recalculated as long as the representativeness number continues to rise with each iteration. When it declines, this process ends. The maximum representativeness number obtained in this way is the one assigned to the user.

Note 3: In various embodiments the discussed calculations are conducted at either the "artist level" or "item level". That is, in some embodiments the artists are rated and calculations done from those ratings and in others item ratings are used.

Appendix B

Introduction

This brief document presents a methodology for clustering songs by calculating "information transfer" as that value is calculated within the framework of Shannon entropy.

First, we will present a simple clustering algorithm will be presented, and second, we will present Python source code for calculating information transfer between clusters and users. Together, these techniques comprise a complete solution for clustering songs.

Clustering Algorithm

For simplicity, and to maximize the probability of showing that our basic approach can find useful clusters, we will use one of the most simple clustering algorithms possible, which does not contain possible optimizations to improve computational speed.

Here are the steps:

```
For each song not yet assigned clusters (note that the first time the system
is started, this would be all songs):
      Randomly assign a cluster.
Repeat:
      For each song (including new songs not yet added to clusters):
            For each cluster other than the original one the song is in:
                  Compute the change in total system information transfer that
                  would occur if the song were moved to the other cluster.

                  If at least one such potential move would result in an
                  increase of information transfer:

                        Execute the move that results in the greatest
                        increase.

      If no movements occurred in the "For each song" loop:

   Delay until there is a new song to process.
```

The above can continue until we want to bring down the system.

It would be great if an administrator console could see, via the Web, a history of the number of distinct songs moved per hour, so that we can monitor how the system is evolving toward stability. If no songs were moved in recent hours, we know that optimization is complete (of course that will only happen if we stop adding songs).

Calculating Information Transfer

A Python example will be used to describe the algorithm.

At the top of the Python listing is a matrix. Each row represents a cluster and each column represents a user. The numbers represent the number of songs in ith cluster that are associated with the jth user. For example the 10^{th} user is associated with 3 songs in the 4^{th} cluster. With Radio Userland data, this would mean that the user has played the song.

When a song is moved from one cluster to another, a number of counts in the matrix may be affected, both in the originating cluster and the target cluster, because that song will be associated with a number of users. Subsequently, the clustering algorithm, which must "try" various possible movements to find the best one, will be very computationally expensive. Various tricks can be used to minimize the number of computations to be done; the Python code below uses virtually no such tricks. It would be appropriate for early Java versions to be equally free of optimizations; for one thing, the fewer optimizations, the less chance for bugs to be introduced into the code. Then we can refine from there, always checking to make sure our optimizations don't change the output numbers. We can check this by loading the database with test data, setting the random number generator to a constant seed, and running the algorithm after each enhancement. The resulting clusterings should always be identical after the same number of iterations. NOTE: There should therefore be some easy way to load the same test data into the system repeatedly.

Obviously, a line-by-line conversion to Java probably doesn't make sense. For one thing, an index-based data structure will probably not be appropriate, because the ID's of the users, after filtering, will not be contiguous. And some users may be dropped from the processing over time for one reason or another. So some kind of map structure would seem to be more appropriate. The row-and-column naming convention would therefore probably also not make sense in the Java version.

Note 1: In the initial release, let's count all user-song-associations as being a 1 no matter how many times the user played the song. So, to get a count of 3 in a cell in the matrix, a user must have played 3 distinct songs. Future versions may count each play as a separate association.

Note 2: it is traditional to use log base 2 when doing Shannon entropy calculations, but if there is no log base 2 function in the Java libraries, we can use natural logarithms.

```
PYTHON CODE BEGINS HERE
```

```
lstLst = [
        [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ]]
lstLst.append(
        [ 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ])
lstLst.append(
        [ 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ])
lstLst.append(
        [ 0, 0, 0, 0, 0, 0, 0, 1, 9, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ])
lstLst.append(
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, ])
lstLst.append(
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, ])
lstLst.append(
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ])


import math

def minusPLogP( float_ ):
        """
                Takes a probability associated with a particular value for a
                random variable, and outputs that value's contribution to the
                Shannon entropy.
        """
        if float_ == 0.0:
            return 0
        else:
                return  - float_ * math.log( float_ )


def countCol():
        """
                The number of users.
        """
        return len( lstLst[ 0 ] )

def countRow():
        """
                Return the number of clusters.
        """
        return len( lstLst )

def sumRow( int_row ):
        """
                Sums user-song-association instances for a single cluster.
        """
        int_sum = 0
        for int_col in range( countCol()):
```

```
            int_sum = int_sum + lstLst[ int_row ][ int_col ]
        return int_sum

def sumCol( int_col ):
        """
            Sums user-song-association instances for a single user.
        """
        int_sum = 0
        for int_row in range( countRow()):
            int_sum = int_sum + lstLst[ int_row ][ int_col ]
        return int_sum

def sumTotal():
        """
            Sums user-song-association instances across the universe.
        """
        int_sum = 0
        for int_row in range( countRow()):
            for int_col in range( countCol()):
                int_sum = int_sum + lstLst[ int_row ][ int_col ]
        return int_sum

def userUncertainty():
        """
            Loop through the users, calculating the probability,
            p, that a randomly
            chosen user-song-association instance would be associated
            with the user being looped through.

            Then sum p log p for all users.

            That is the Shannon uncertainty for the user population.
        """

        float_sum = 0.0
        int_total = sumTotal()
        for int_col in range( countCol()):
            float_p = float( sumCol( int_col )) / int_total
            float_sum = float_sum + minusPLogP( float_p )
        return float_sum

def clusterUncertainty():
        """
            Loop through the clusters, calculating the probability, p,
            that a randomly
            chosen user-song-association instance would be associated
            with the cluster being looped through.
```

```
            Then sum p log p for all clusters.

            That is the Shannon uncertainty for the cluster population.
        """
        float_sum = 0.0
        int_total = sumTotal()
        for int_row in range( countRow()):
            float_p = float( sumRow( int_row )) / int_total
            float_sum = float_sum + minusPLogP( float_p )
        return float_sum


def jointUncertainty():
        """
            Loop through all unique combinations of user-cluster,
            calculating the probability, p, that a randomly
            chosen user-song-association instance would be associated
            with the user-cluster combination being looped through.

            Then sum p log p for all user-cluster combinations.

            That is the joint Shannon uncertainty for the cluster population.
        """
        float_sum = 0.0
        int_total = sumTotal()
        for int_row in range( countRow()):
            for int_col in range( countCol()):
                float_p = float( lstLst[ int_row ][ int_col ]) / int_total
                float_sum = float_sum + minusPLogP( float_p )
        return float_sum

def calculateInformationTransfer():
        """
            Calculate the information transfer.
        """
        return userUncertainty() + clusterUncertainty() - jointUncertainty()

print 'User uncertainty: ', userUncertainty()
print 'Cluster uncertainty: ', clusterUncertainty()
print 'Joint uncertainty: ', jointUncertainty()
print 'Information transfer: ', calculateInformationTransfer()
```

An Optimization Strategy

This strategy is used by preferred embodiments of the information transfer algorithm.